Лекция 13. Структуры и другие формы данных. Часть 2.

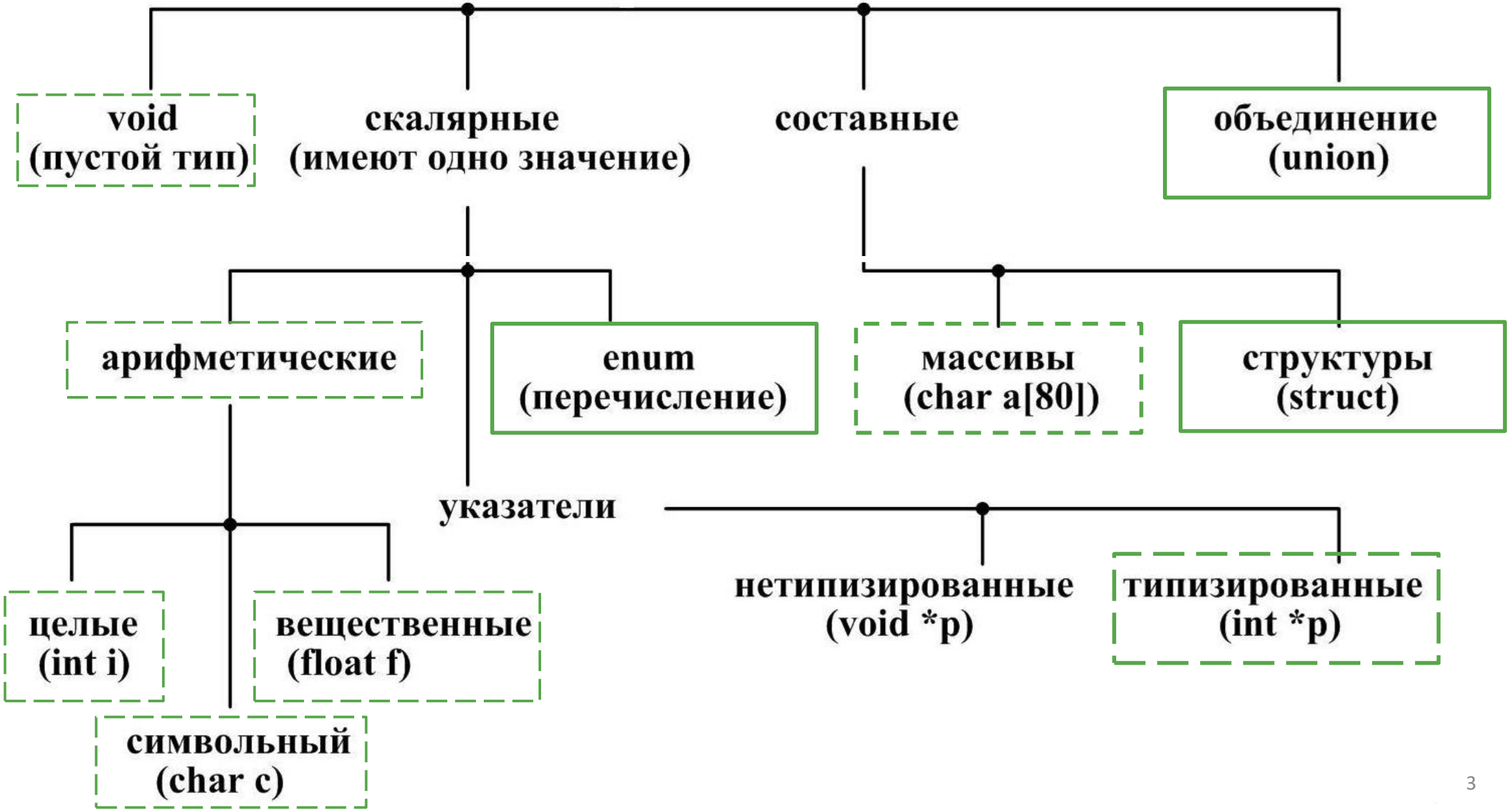
Ревун Артем Леонидович
ст. преп. кафедры вычислительных систем

Курс «Программирование», осенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)

Содержание

- Ключевые слова: union, typedef
- Структуры в языке Си способы создания шаблонов и переменных типа структур
- Доступ к членам структуры и написание функций для обработки структур
- Средство typedef в языке Си
- Объединения и указатели на функции

Типы данных в Си (прогресс)



Новые подходы в представлении данных

Представление данных один из важнейших этапов при разработке программы.

Во многих случаях простой *переменной* или даже *массива* оказывается недостаточно. Язык **Си** позволяет расширить возможности *представления данных* с помощью *переменных* типа **структур**.

В своей базовой форме **структура Си** является достаточно гибким средством, чтобы представлять широкое разнообразие данных, и она позволяет изобретать новые формы *представления данных*.

Массив структур (1)

```
struct accoun{
    int id;
     struct {char first[20]; char
last[20];};
     double money;
};
#define SIZE 5
double allMoney (const struct account
                 bank[],   int size);
```

Массив структур (2)

```
double allMoney(const struct account
bank[],                          int size) {
    double result = 0.;
    for (int i = 0; i < size; i++)
    {
        result+=bank[i].money;
    }
    return result;
}
```

Массив структур (3)

```
struct account bank[SIZE] = {
{1, {"Richard","Murray"}, 183.81},
{2, {"Michael","Brown"}, 988.09},
{3, {"Barbara","Spencer"}, 990.31},
{4, {"Mark","Jackson"}, 309.89},
{5, {"Jack","Green"}, 121.18}
};
printf("На счёте банка суммарно: %lf$
\n",   allMoney(bank, SIZE));
```

Массив структур (4)

- Имя массива можно использовать для передачи в функцию адреса первой структуры массива.
- Для доступа к последующим структурам массива можно применять запись с квадратными скобками. Обратите внимание, что вызов функции `allMoney(bank, SIZE)` приведет к таким же результатам, как и в случае указания имени массива, поскольку `bank`и `&bank[0]`— это один и тот же адрес. Использование имени массива представляет собой просто косвенный способ передачи адреса структуры.
- Из-за того, что функция `allMoney()` не должна изменять исходные данные, в ней применяется квалификатор `const` из ANSI Си.

Объединения (1)

Объединения — это тип, который позволяет хранить данные разных типов в одном и том же месте памяти (но не одновременно). Типичным видом объединения может служить таблица, предназначенная для хранения смеси типов в определенном порядке, который не является ни регулярным, ни известным заранее. Применяя массив объединений, можно создать массив единиц одинаковых размеров, каждая из которых может содержать данные разнообразных типов.

Объединения (2)

```
union hold {
    int digit;
    double bigfl;
    char letter;
};
union hold fit = {1.00};
union hold save[10] =
   {1,2,3,4,5,6,7,8,9,10};
union hold  qu = {'Q'};
```

Объединения (3)

Объединения можно инициализировать.

```
union hold {
    int digit;
    double bigfl;
    char letter;
};
union hold valA;
valA.letter = 'R ';
union hold valB = valA;
union hold valD = {.bigfl = 118.2};
```

Объединения (4)

Оператор (.) покажет какой тип данных применяется в текущий момент. За один раз сохраняется только одно значение. Нельзя одновременно хранить значение **char** и **int**. Ответственность за отслеживание в программе типа данных, хранящегося в текущий момент внутри объединения, возлагается на вас.

```
fit.digit = 23; // хранится 23; 2 bytes
fit.bigfl =2.0; // 23 очищено, 2.0 сохранено;
8 bytes
fit.letter = 'h'; // 2.0 очищен, h сохранено;
1 byte
```

Объединения (5)

Вы можете использовать **операцию ->** с указателями на объединения в той же манере, как применяли ее с указателями на структуры:

```
union hold {
    int digit;
    double bigfl;
    char letter;
}; union hold fit;
   union hold  * pu = &fit;
   int x = pu->digit;
```

Объединения (6)

```
union hold {
    int digit;
    double bigfl;
    char letter;
};
union  hold fit;
    fit.letter = 'A';
    double flnum = 3.02*fit.bigfl;
    printf("%lf", flnum); // 0.000000
```

Объединения (7)

Польза применения объединений заключается в оправданных операциях перестановки.

```
struct owner {
char socsecurity[12]; …};
struct leasecompany {
char name[40];
char headquarters[40];…};
union data {
```

Объединения (8)

```
struct owner owncar;
struct leasecompany leasecar;};
struct car data {
    char make [15];
    int status;
    union data ownerinfo;
};
```

Если значение `status` равно 0, программа может использовать `ownerinfo.owncar`, а если значение `status` равно 0 - то `ownerinfo.leasecar`.

Объединения (8)

```
struct car data {
    char make [15] ;
    int status; /* 0 = принадлежит, 1 =
взят напрокат */
    union {
        struct owner owncar;
        sstruct leasecompany leasecar;
        }; По аналогии с анонимными
    ... структурами можно использовать
};      анонимные объединения.
```

Перечисления (1)

Перечисления - тип можно использовать для объявления символических имен, представляющих целочисленные константы. Ключевое слово **enum** позволяет создать новый “тип” и указать значения, которые для него допускаются. Константы **enum** имеют тип **int**, поэтому их можно применять везде, где разрешено использовать тип **int**.) Целью перечислимых типов является улучшение читабельности программы.

```
enum spectrum {red, orange, yellow,
green, blue, violet};
enum spectrum color;
```

Перечисления (2)

Таким образом, возможными значениями color будут `red, orange, yellow…`

Эти символические константы называются перечислителями.

```
int c; color = blue;
if (color == yellow)
…
for(color=red;color<=violet;color++)
…
```

Например, перечислимые константы для `spectrum` входят в диапазон 0-5, так что для представления переменной `color` компилятор мог бы выбрать тип **unsigned char**.

Перечисления (3)

Стандартные значения

```
enum kids {nippy, slats, skippy, nina, liz};
```

0 1 2 3 4

Присвоенные значения

```
enum levels {low = 100, medium = 500, high =
2000};
```

Перечисления (4)

```
…
switch(color)
{
  case red : puts("Розы красные.");break;
  case orange : puts("Маки оранжевые.");break;
  case yellow : puts("Подсолнухи желтые.");break;
  case green : puts("Трава зеленая.");break;
  case blue : puts ("Колокольчики синие.");break;
  case violet : puts("Фиалки фиолетовые.") ;break;
 default: printf("Цвет %s не известен.\n", color);
}
…
```

Перечисления (4)

```
…
switch(color)
{
  case red : puts("Розы красные.");break;
  case orange : puts("Маки оранжевые.");break;
  case yellow : puts("Подсолнухи желтые.");break;
  case green : puts("Трава зеленая.");break;
  case blue : puts ("Колокольчики синие.");break;
  case violet : puts("Фиалки фиолетовые.") ;break;
 default: printf("Цвет %s не известен.\n", color);
}
…
```

Пространства имён (1)

Термин ***пространство имен*** в языке Си применяется для идентификации частей программы, в которых распознается то или иное ***имя***. ***Область видимости*** входит в состав этой концепции: две переменные, имеющие одно и то же имя, но разные области видимости, не конфликтуют друг с другом, в отличие от двух переменных с одинаковыми именами и одной и той же областью видимости.

```
struct rect a = {10, 20};  int rect = 59;
int a=10; int a=20;
```

Пространства имён (2)

Возможность `typedef` представляет собой усовершенствованное средство манипулирования данными, которое позволяет создавать собственное имя для ***типа данных***.

В этом отношении оно подобно директиве `#define`, но с тремя отличиями.

- В отличие от `#define`, средство `typedef` ограничено назначением символических имен только типам, но не значениям.
- Интерпретация `typedef` выполняется компилятором, а не препроцессором.
- В рамках своих ограничений средство `typedef` является более гибким, чем `#define.`

Пространства имён (3)

Предположим , что вы хотите использовать элемент BYTE для обозначения однобайтовых чисел.

```
typedef unsigned char BYTE;
```

После этого BYTE можно применять для определения переменных:

```
BYTE x, y[10], * z;
```

Пространства имён (4)

Область видимости этого определения зависит от места расположения оператора `typedef`.

Локальное объявление (внутри функции)

```
void showByte(){
    typedef unsigned char BYTE;
    ...
}

BYTE y[10];
```

Пространства имён (5)

Область видимости этого определения зависит от места расположения оператора `typedef`.

Глобальное объявление:

```
typedef unsigned char BYTE;
void showByte(){
        BYTE x;√

    ...
}
BYTE y[10];√
```

Пространства имён (6)

Для определений `typedef` часто используются прописные буквы, чтобы напоминать пользователю о том, что имя типа в действительности является **символическим сокращением**, но разрешены также и строчные буквы:

```
typedef unsigned char byte;
```

ИЛИ

```
typedef unsigned char BYTE;
```

Регламент именования соответствует именованию переменных!!!

Пространства имён (7)

typedef позволяет то что не может #define.

```
typedef char * STRING;
STRING name, sign;
//char * name, * sign;

#define STRING char *
STRING name, sign;
//char * name, sign;
```

Пространства имён (7)

Можно также использовать со структурами:

```
typedef struct complex {
    float real;
    float imag;
} COMPLEX;
```

Теперь для представления комплексных чисел вместо структуры по имени `complex` можно применять тип `COMPLEX`.

Сложные формы данных (1)

Благодаря *структурам*, *объединениям* и `typedef`, язык Си предоставляет инструменты для эффективной и переносимой обработки данных. Когда вы делаете объявление, ***имя*** (или ***идентификатор***) можно изменить, добавив модификатор.

Модификатор	Описание
*	Обозначает указатель
()	Обозначает функцию
[]	Обозначает массив

Сложные формы данных (2)

Оператор [], которые обозначают массив, и **оператор ()**, обозначающие функцию, имеют одинаковый приоритет. Этот приоритет *выше* , чем у **операции разыменования ***, которая означает, что следующее объявление делает `riskv` массивом указателей, а не указателем на массив:

```
int * riskv[10];
```

Модификатор	Описание
*	Обозначает указатель
()	Обозначает функцию
[]	Обозначает массив

Сложные формы данных (3)

Операторы [] и () имеют *ассоциативность слева направо*. Таким образом, приведенное ниже объявление делает `goods` массивом из 12 массивов, содержащих по 50 значений int, а не массивом из 50 массивов с 12 элементами типа int:

```
int goods[12][50];
```

Модификатор	Описание
*	Обозначает указатель
()	Обозначает функцию
[]	Обозначает массив

Сложные формы данных (4)

Операторы [] и () имеют один и тот же приоритет, но из-за их *ассоциативности слева направо* в следующем объявлении * и `riskv` группируются вместе перед применением **оператора []**. Это делает `riskv` указателем на массив из 10 значений int:

```
int (* riskv)[10];
```

Модификатор	Описание
*	Обозначает указатель
()	Обозначает функцию
[]	Обозначает массив

Сложные формы данных (5)

```
int board[8][8];
// массив из массивов значений int
int **ptr;
// указатель на указатель на int
int *risks[10];
// 10-элементный массив указателей на int
int (*rusks) [10];
// указатель на массив из 10 значений int
```

Сложные формы данных (6)

```
int *oof [3][4];
// массив размером 3 x 4 указателей на
int
int (*uuf)[3][4];
// указатель на массив размером 3 x 4
значений int
int (*uof[3])[4];
// 3-элементный массив указателей на 4-
элементные массивы значений int
```

Сложные формы данных (7)

```
char * fump(int); // функция,
возвращающая указатель на char
char (* frump)(int); // указатель
на функцию, возвращающую тип char
char (* flump[3]) (int); // массив
из 3 указателей на функции,
которые
```

Сложные формы данных (8)

```
typedef int arr5[5];
typedef arr5 * p_arr5;
typedef p_arr5 arrp10[10];
arr5 togs; // togs - массив из 5 значений
int
p_arr5 p2; // p2 - указатель на массив из
5 значений int
arrp10 ap; // ap - массив, содержащий 10
указателей на массивы из 5 значений int
```

Сложные формы данных и функции (1)

Язык Си допускает объявление указателей на функции.
Обычно указатель на функцию используется в качестве аргумента в другой функции, сообщая ей, какую функцию применять.
Функция qsort() из библиотеки Си (***stdlib.h***) спроектирована на работу с массивами любого вида при условии, что вы уведомите ее о том, какую функцию применять для сравнения элементов.

Сложные формы данных и функции (2)

```
#include <stdlib.h>
int values[] = { 40, 10, 100, 90, 20, 25 };
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}
…
qsort (values, 6, sizeof(int), compare);
for (int n=0; n<6; n++)
    printf ("%d ",values[n]);
```

Сложные формы данных и функции (2)

```
#include <stdlib.h>
int values[] = { 40, 10, 100, 90, 20, 25 };
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}
…
qsort (values, 6, sizeof(int), compare);
for (int n=0; n<6; n++)
    printf ("%d ",values[n]);
```

Задачи на практику

Читать:

1. Глава 14 *Структуры и другие формы данных* (стр. 601 дочитать).

Изучить:

1. *Функции и указатели* (стр. 612)

Готовится к экзамену.

Спасибо за внимание

Ревун Артем Леонидович
ст. преп. Кафедры вычислительных систем

Курс «Программирование», осенний семестр, 2024
Сибирский государственный университет телекоммуникаций и информатики (г. Новосибирск)